# Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

## ABSTRACT

## ACCELERATING TRANSITIVE CLOSURE OF LARGE-SCALE SPARSE GRAPHS

**by**
**Sanyamee Milindkumar Patel**

Finding the transitive closure of a graph is a fundamental graph problem where another graph is obtained in which an edge exists between two nodes if and only if there is a path in our graph from one node to the other. The reachability matrix of a graph is its transitive closure. This thesis describes a novel approach that uses anti-sections to obtain the transitive closure of a graph. It also examines its advantages when implemented in parallel on a GPU using the Hornet graph data structure.

Graph representations of real-world systems are typically sparse in nature due to lesser connectivity between nodes. The anti-section approach is designed specifically to improve performance for large scale sparse graphs. The NVIDIA Titan V GPU is used for the execution of the anti-section parallel implementations. The Dual-Round and Hash-Based implementations of the Anti-Section transitive closure approach provide a significant speedup over several parallel and sequential implementations.

# ACCELERATING TRANSITIVE CLOSURE OF LARGE-SCALE SPARSE GRAPHS

by
Sanyamee Milindkumar Patel

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science

December 2020

**APPROVAL PAGE**

**ACCELERATING TRANSITIVE CLOSURE OF LARGE-SCALE SPARSE GRAPHS**

**Sanyamee Milindkumar Patel**

---

Dr. David A. Bader, Thesis Advisor — Date
Distinguished Professor, New Jersey Institute of Technology

---

Dr. Ioannis Koutis, Committee Member — Date
Associate Professor, New Jersey Institute of Technology

---

Dr. Dimitrios Theodoratos, Committee Member — Date
Associate Professor, New Jersey Institute of Technology

## BIOGRAPHICAL SKETCH

**Author:** Sanyamee Milindkumar Patel

**Degree:** Master of Science

**Date:** December 2020

### Undergraduate and Graduate Education:

- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2020
- Bachelor of Technology in Computer Engineering,
  Narsee Monjee Institute of Management Studies, Mumbai, Maharashtra, India, 2018

**Major:** Computer Science

*Today many fields consider network science their own. Mathematicians rightly claim ownership and priority through graph theory; the exploration of social networks by sociologists goes back decades; physics lent the universality concept and infused many analytical tools that are now unavoidable in the study of networks; biology invested hundreds of millions of dollars into mapping subcellular networks; computer science offered an algorithmic perspective, allowing us to explore very large networks; engineering invested considerable efforts into the exploration of infrastructural networks. It is remarkable how these many disparate pieces managed to fit together, giving birth to a new discipline.*

Albert-László Barabási

## ACKNOWLEDGMENT

I would like to thank Dr. David Bader for accepting to be my Thesis Advisor and being supportive even when I faltered, at every step in the process. Dr. Bader, I am incredibly grateful to have started my journey in research with you. Even if the pandemic made things tougher, you were always there for helping me in this journey. I am thankful to Dr. Ioannis Koutis and Dr. Dimitrios Theodoratos from NJIT for agreeing to be part of my thesis committee, for their valuable input and for playing a huge part in shaping the course of graduate education. I would like to thank Dr. Oded Green from NVIDIA and Dr. Zhihui Du from the New Jersey Institute of Technology for working tirelessly for the completion of this project and their dedication to make the best version of this project possible. Without their contribution, I would not be writing this paper. I would also like to thank Dr. Hang Liu and Zehui Xie from Stevens Institute of Technology. It was an absolute pleasure to learn from them and work with them. Finally, I would like to thank my parents, friends and fellow students at NJIT who created a conducive environment for me in tough times like these.

## TABLE OF CONTENTS

## TABLE OF CONTENTS
## (Continued)

**Chapter** **Page**

## LIST OF TABLES

**Table** **Page**

## LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

Real-world systems where connections are fundamental in their functioning are best represented by graphs. These are systems where components either interact with each other or are related to each other based on the state of the system. Numerous problems in various domains can be represented as graphs. In a graph, the number of possible edges from a single node can be equal to the total number of nodes in the graph. For directed graphs, the total number of possible edges in a graph with N nodes is $N \times (N - 1)$. For undirected graphs, the number is half of that since directed graph has the possibility of two edges between two same nodes. Graphs which utilize this property are known as dense graphs. These graphs have a larger number of edges. Theoretically, basic data structure and algorithms for graphs were constructed keeping the dense nature of graphs in consideration. However, real-world systems, when represented as graphs, have a significantly lesser number of edges [14]. Networks where nodes have a small fraction of the possible edges are commonly referred to as sparse networks.

The transitive closure [16] of a directed graph is defined as another graph in which an edge exists between two nodes, A and B, if and only if there is a path between A to B in the graph. Given a directed graph, G(V,E) where V is the set of vertices and E is the set of edges in G, the transitive closure, G' of G can be defined as G'(V,E') where E' contains edges (i,j) if and only if there is a directed path from vertex i to vertex j in E. In simpler words, the transitive closure of a graph depicts all the node-pairs which have a path from one to the other in the original graph. An edge from node i to node j in the transitive closure of a graph G implies that there is a directed path from node to to node j in G.

The problem of transitive closure is represented in various forms. The single source transitive closure problem requires all paths that are reachable from a given source vertex[5]. A depth-first search from the source vertex is an algorithm for the problem with $O(|V|+|E|)$ run-time complexity. A multi-source transitive closure version requires the same answers as single course transitive closure but with multiple source vertices. In the all-pairs transitive closure problem, all pairs of nodes that have a from one to the other should have a respective edge in the transitive closure. An all-pairs shortest path problem further examines the paths represented by edges in the all-pairs transitive closure of a graph. We examine the all-pairs transitive closure problem and refer to it as transitive closure.

Transitive closure is fundamental in solving reachability problems for database querying and in reachability analysis of transition networks in distributed systems [10]. If a transitive closure of a relation is computed prior to querying and saved, there is scope for rapid evaluation of transitive queries [27, 1]. Transitive closure also finds its applications in compiler construction for parsing automata [5, 43]. Real-world graphs being large and sparse in nature increases the scope for customization in traditional transitive closure algorithms to reduce the time and space complexity by leveraging the power of parallel programming paradigms. Advantages of the properties of directed graphs and sparse graphs can be taken by improving methods for representation of graphs and their processing. We examine the full transitive closure problem for directed sparse graphs of larger sizes using GPUs.

## 1.1 Goals

Our primary goal was to find an algorithm for transitive closure specifically for large scale sparse graphs using the Hornet data structure representation of graphs. We searched for transitive closure algorithms that require a time that is almost linear to the size of the input graph in the average case. We utilized the dynamic graph data

structure Hornet. It provides fast edge insertions and efficient memory management, helping the algorithm to outperform CPU and GPU implementations of the algorithm.

The secondary goal was to parallelize the algorithm and optimize the Anti-Section approach using different programming paradigms for scalability. We designed a Dual-Round version and a hashing version of our Anti-Section approach. We estimated their performance and identified drawbacks to iteratively improve both approaches.

## 1.2 Contributions

In this thesis, we introduce a new algorithm for obtaining the transitive closure of a graph called the Anti-Section Transitive Closure. Most of the credit for the conception and implementation of the work go to Dr. Oded Green from NVIDIA and Dr. Zhihui Du from NJIT who are esteemed members of Dr. David A. Bader's research group. Following are the major contributions of the paper to algorithm.

- An operator to obtain the non-intersecting elements of two lists called the list Anti-Section operator. The concept is the opposite of finding an intersection of two lists. Hence, it is called Anti-Section.

- We employ a bulk edge insertion technique in our algorithm by utilizing Hornet's dynamic edge insertion advantage. This helps us avoid complex locking mechanisms for each iteration of finding new edges using the Anti-Section approach. Using Hornet also reduces memory utilization.

- We implement a parallelized version of the Anti-Section Transitive Closure algorithm showing effectiveness of load balancing across tens of thousands of threads.

- The Dual-Round and Hash-Based Anti-Section algorithms outrun various CPU and GPU implementations of transitive closure.

## 1.3 Thesis Outline

In Chapter 2, we define the graph theoretic terminology that we use in the thesis, and the transitive closure problem and discuss its variations. We delve in to previous solutions of the problem and work that is relevant to the problem specifically for large scale sparse graphs.

In Chapter 3, we delve into details about the Hornet graph data structure and provide reasoning for its choice as the data structure for graph representation for this problem.

In Chapter 4, we study the Anti-Section Transitive Closure algorithm. We start with describing the triangle counting algorithm to understand the relevance of "anti-sections". We explore the logic for an Anti-Section in a graph and the iteration-based method using Hornet to gain an understanding of the need for two different implementations of Anti-Section transitive closure. Then, we develop and evaluate the Dual-Round and Hash-Based approaches of the algorithm, and analyze the time and space complexity of their implementations.

In Chapter 5, we present the experimental setup used to measure performance of both implementations of our algorithm. This consists of the input graph networks, software, hardware environments and frameworks used as benchmarks, the CPU and GPU implementations of Transitive Closure to compare with our algorithm.

In Chapter 6, we present the performance analysis of the Anti-Section Transitive Closure algorithm over various aspects in comparison with the CPU and GPU implementations explained in Chapter 5.

In Chapter 7, we present the conclusions of the thesis and future scope of the work.

# CHAPTER 2

# THE TRANSITIVE CLOSURE PROBLEM

## 2.1 Graph Terminology

**Definition 1.** A directed graph G is a tuple $(V, E)$ where V is a set of elements called vertices and $E \subseteq V \times V$ is set of ordered pairs called edges. The cardinality of V is denoted by n and the cardinality of E is denoted by e. Given an edge $(u, v)$, u is the source and v is the destination of the edge.

In this thesis, we consider transitive closure only for directed graphs. Therefore, the word "graph" always refers to a directed graph, from here on, in this thesis. An example of a directed graph is depicted in Fig. 2.1. Note that the arrow depicting an edge starts from the source node to the destination node.

Another term that will frequently be used in this thesis is sparse graphs. There is no set threshold for considering a graph as sparse. The maximum possible number of edges in a directed graph $G(V, E)$, is $|V|(|V| - 1)$. This is approximately equal to the square of the total number of vertices in G i.e $|V|^2$. A graph that has a significantly lesser number of edges than $|V|(|V| - 1)$ is considered as sparse. The number of edges in a sparse graph is $O(|V|)$ which is the minimal number of edges in a graph.

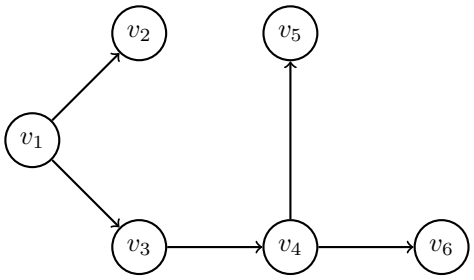



**Figure 2.1** An example graph, $G(V, E)$.

$$
\begin{array}{c}
\\
v_1 \\
v_2 \\
v_3 \\
v_4 \\
v_5 \\
v_6
\end{array}
\begin{array}{c}
\begin{array}{cccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \end{array} \\
\left( \begin{array}{cccccc}
0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{array} \right)
\end{array}
$$

**(a)** Adjacency Matrix

$$
\begin{array}{lcll}
[v_1] & = & v_2, & v_3 \\
[v_2] & = & \times & \\
[v_3] & = & v_4, & \\
[v_4] & = & v_5, & v_6 \\
[v_5] & = & \times & \\
[v_6] & = & \times &
\end{array}
$$

**(b)** Adjacency List

**Figure 2.2** Representation of G(V,E) using basic data structures.

### 2.1.1 Graph Representation

The choice of data structure to represent a graph can depend on the properties of the graph, the system being modelled as a graph and the type of problem that has to be tackled using the graph. A graph can be represented using a number of data structures such as adjacency matrix and adjacency list [4]. Fig. 2.2 depicts both representations of $G(V, E)$. An adjacency matrix is a $|V| \times |V|$ matrix of bits where element $(i, j)$ is 1 if and only if the edge $(vi, vj)$ is in E. Thus an adjacency matrix takes up $\Theta(|V|^2)$ storage. In Fig. 2.2(a) depicting the adjacency matrix of $G(V, E)$, the number of zero-values denoting the absence of an edge between two nodes are many. An adjacency list is a list of $|V|$ lists, each list corresponding to a vertex in the graph containing the neighbors of that vertex. It requires $\Theta(|V| + |E|)$ space. Due to a sparse graph containing many redundant zero values, adjacency list is a more suitable choice. However, in the case of sparse graphs, each node may not have neighboring edges to which they are connected. This is true for nodes $v_2$, $v_5$, and $v_6$ in $G(V, E)$ in 2.2.

Another aspect to consider is the time and space complexity of conducting basic graph operations using a particular representation. While an adjacency matrix utilizes $O(|V|^2)$ space, it offers O(1) i.e. constant time lookup. On the other hand,

an adjacency list requires $\Theta(|V| + |E|)$ space but needs $O(|V|)$ time for lookup. Furthermore, the set of adjacent vertices to a particular vertex can be accessed quicker through an adjacency list than an adjacency matrix. It will be $O(|V|)$ for the adjacency matrix. For the adjacency list, finding the list of the vertex is all that needs to be done. Traversing the adjacency list of that vertex will give us the entire set in $O(|neighbors|)$ time. For solutions to scale for larger and sparser graphs, a balance between space and time trade-off needs to be achieved. Therefore, the need for more efficient graph data structures to represent larger sparse graphs is evident.

Graph representations such as CSR (Compressed Sparse Row), CSC (Compressed Sparse Column) and COO (Co-ordinate List) are used for sparse graphs although the most commonly used implementation for simpler problems is adjacency list. The concept behind these structures is to reduce storage and computing time by recording and traversing only the non-zero elements. The most widely used one is the compressed sparse row (CSR) format. It stores only the column indices and values of non-zero elements existing in a row. The start and end of each row are then stored as column indices and value in a row offsets array. Hence, CSR only requires $O(|V|+|E|)$ memory for storage. Various other representations have been developed and can be found in [15, 22, 48, 31, 41]. All these structures have trade-offs in terms of space and time complexity for lookup and manipulation of graphs. For our principal approach, we employ Hornet[11], a data structure specifically designed for large scale sparse graphs. More information regarding Hornet and the advantages it provides to our implementation is given in Chapter 3.

## 2.2 The Transitive Closure Problem

**Definition 2.** The transitive closure of a directed graph $G(V, E)$ is another graph $\hat{G}(V, \hat{E})$ with edge $(v, w)$ in $\hat{G}$ if and only if there is a path from $v$ to $w$ in G.

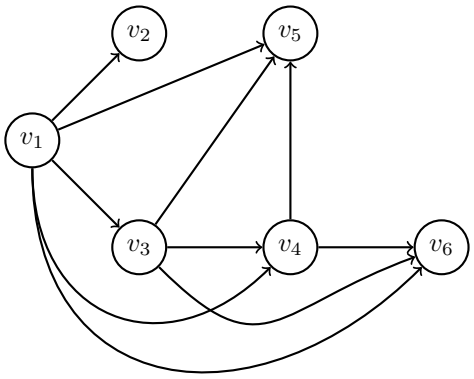



**Figure 2.3** Transitive Closure $\hat{G}(V, \hat{E})$ of example graph, $G(V, E)$.

For the graph depicted in 2.1, the transitive closure can be calculated as the graph depicted in 2.3. $\hat{G}$ denotes the transitive closure of G and $\hat{E}$ denotes the set of edges in the transitive closure of E. One example of a new edge added in the transitive closure is the edge from $v_1$ to $v_4$. This is added due to edges $(v_1, v_3)$ and $(v_3, v_4)$ resulting in a path from $v_1$ to $v_4$.

The property of transitivity is known in many disciplines such as mathematics and database systems. It means that if an object $A$ is related to an object $B$, and object $B$ is related to another object $C$, then by the property of transitivity, $A$ is related to $B$.

A major challenge in transitive closure algorithms is avoiding redundant operations. In a single graph, two nodes may be connected through many paths. This results in exploring each path and trying to insert the seemingly new edge to the transitive closure of the graph. This also means that successors of connected nodes will intersect. We aim to efficiently find these intersections. We also avoid the bottleneck caused by insertion of bulk edges at every iteration with the properties of Hornet.

The transitive closure problem is an instance of the closed semiring problem in linear algebra. Of the various algorithms that can be designed to solve closed

semiring problems, implementations specific to graph properties and the requirements of the problem have are very few. We discuss relevant works in this area in further subsections. However, a lot of these implementations are designed for the general case and do not apply to transitive closure algorithms for sparse graphs. We discuss a more relevant approach in the subsequent subsection. Transitive closure algorithms have also been studied in the area of databases to implement efficient querying methods for transitive relationships [3, 45, 42].

## 2.3 Previous Work

Transitive closure has encouraged various themes in approaching the problem in terms of algorithm and implementation. Research on this topic has been broad over the years. Transitive closure is also closely associated with the all-pairs shortest paths problem where shortest paths between all vertex pairs are obtained. For obtaining the transitive closure of a graph, we need to explore all possible paths between all vertex pairs in the worst case. Using the brute-force approach, transitive closure can be solved by conducting a graph traversal for each vertex in the graph. If a vertex is reached then the corresponding vertex is indicated to have a path with the originating vertex. Assuming that the graph was represented by an adjacency matrix, then the cost is $\Theta(n^3)$ where n is the number of vertices in the graph. For a sparse graph, an adjacency list is more appropriate, then the cost is $\Theta((n + m)n)$. We present approaches to generate transitive closure by bifurcating them into groups of algorithms based on their principal underlying logic.

### 2.3.1 Warshall's Algorithm

The first group, based mainly on Boolean matrix multiplication was pioneered by Roy in [40] but it only found recognition later when it was rediscovered by Warshall in [47]. Warshall's transitive closure algorithm generates a sequence of n matrices to

calculate the transitive closure, where n is the number of vertices in the graph, as follows.

$$R_0,\ R_1,\ R_2....R_{k-1},\ R_k,....R_n$$

In the above sequence, the $R_0$ matrix represents paths without any intermediate vertices, so it is the adjacency matrix of the original graph. In any matrix $R_k$, the value of the element at the $i^{th}$ row and $j^{th}$ column is one if and only if there exists a path from $v_i$ to $v_j$ such that all intermediate vertices, $w_q$ are among the first $k$ vertices. This means that, at any intermediate step, $k$, the paths discovered are only up to the first k vertices in the graph. The $R_n$ matrix represents the transitive closure and has ones if there is a path between vertices to any of the n vertices in the graph.

**Algorithm 1** Warshall's Algorithm

```
1: procedure WARSHALL(A[1...n][1...n])
2:     R_0 ← A
3:     for k ← 1, n do
4:         for i ← 1, n do
5:             for j ← 1, n do
6:                 R_k[i, j] ← R_{k-1}[i, j] OR (R_{k-1}[i, k] AND R_{k-1}[k, j])
7:             end for
8:         end for
9:     end for
10: end procedure
```

Illustrated in Operation 6 of Algorithm 1, the matrix being constructed for every $k^{th}$ iteration is populated on the basis of the presence of paths between the $[i, k]$ and $[k, j]$ demonstrating the transitive property explicitly. The worst case for this algorithm is $O(n^3)$. Moreover, the $R_k$ matrix needs to be stored for every iteration.

For the case of sparse graphs, Warshall's algorithm performs worse than the brute-force method. Hence, we do not consider it for optimization.

An algorithm with a better execution time but a worst-case complexity of $O(n^3)$ was proposed by Warren in [46]. It is a two-pass algorithm as opposed to Warshall's one-pass approach. As illustrated in Algorithm 1, each pass of Warren's algorithm goes over only half of the graph and scans the adjacency matrix by rows. One pass scans all edges in the lower triangular portion of the matrix below of the main diagonal and updates each row using it's preceding rows. In the second pass, the upper triangular matrix above the main diagonal is scanned and rows are updates based on the succeeding rows. The principal advantage of Warren's algorithm is that it is more memory efficient and can be applicable when the matrix is too large to fit in memory. However, the time complexity drawback outweighs the space efficiency advantage for our case. Both algorithms scan and modify the values of the adjacency matrix but in a different order.

### 2.3.2 Boolean Matrix Multiplication

Boolean matrix multiplication is closely associated with generating transitive closure. Let us say graph $G(V, E)$ is represented by an $n \times n$ adjacency matrix A that contains Boolean values for respective A[i,j] elements depending on whether an edge exists between vertices $V_i$ and $V_j$. Then $A^2 = A \wedge A$ is another $n \times n$ Boolean matrix such that $A^2[i, j] =$ **true** if and only if exists is a path of length 2 in G from $V_i$ to $V_j$. For a transitive closured edge value to be **true**, the path length between $V_i$ and $V_j$ can be of any length less than $n$. Therefore, transitive closure, $A^*$ of $A$ can be calculated as follows.

$A^* = A \vee A^2 \vee A^3 \vee ...A^n$

Thus, the time complexity of computing the transitive closure of an n node graph is equivalent to that of multiplying two Boolean $n \times n$ matrices n times [16, 33]. O'

Neil *et al.* remark in [36] that if the Boolean product of two $n \times n$ matrices can be calculated in $O(n^{\beta})$ elementary operations, then it should take $O(n^{\beta}.log_2 n)$ time to compute the transitive closure of any $n \times n$ Boolean matrix. For instance, by applying the fast matrix multiplication algorithms such as Strassen's algorithm which runs in $O(n^{log_2 7})$ time. Therefore, as indicated in [36] we can utilize the advantage of Strassen's method to generate transitive closure in $\Theta(n^2.81)$ time. Thus, the best known transitive closure algorithm in terms of $n$ is $O(n^w)$ where w is the width or diameter of the graph.

The Four Russians' algorithm [6] works on semirings directly, and has complexity $O(n^3/logn)$, requiring cubic time for transitive closure computation. Following from the Four Russians' algorithm, Blelloch *et al.* [9] propose a combinatorial data structure to accelerate sparse matrix multiplications giving a lower bound for transitive closure.

Although, matrix multiplication algorithms have been proved to have lower asymptotic bounds, they require processing elements in a particular order. This might prove harmful for parallel algorithms. Moreover, the use of adjacency matrix for representing sparse graphs makes them impractical for the use of large scale sparse graphs. We study methods specific to sparse graphs in detail in Subsection 2.3.4.

### 2.3.3 Special Graph Properties and Strongly Connected Components

Another group of algorithms is based on generating the transitive closure by utilizing specific properties of the graph structure [35]. Strongly connected components in a graph are defined as subset of vertices such that there is a path between each pair of vertices belonging to that subset. This automatically helps us derive the transitive closured edges such that if two vertices $v_1$ and $v_2$ belong to a strongly connected component, $(v_1, v_2)$ and $(v_2, v_1)$ are added in the transitive closure.

The first transitive closure algorithm based on strongly connected components was presented by Purdom [39]. Purdom's algorithm follows Tarjan's algorithm [44] to obtain strongly connected components of a graph, G. A condensation graph, $\bar{G}$ is obtained by representing each component by a single node. The components in $\bar{G}$ are sorted topologically. Then, the transitive closure of the $\bar{G}$ graph is obtained starting from the higher indexed vertices to ensure that vertices appearing later are all included in the successor sets of early appearing vertices. Then the transitive closure of G is generated from the $\bar{G}$. The time complexity of this algorithm is $O(|E| + \mu|V|)$ where $\mu$ is the number of strongly connected components of the graph. This also gives inspiration for finding the transitive closure of a directed acyclic graph by topologically sorting and calculating the transitive closure backwards. Munro [33] proposed an improvement in Purdom's approach by maintaining the information of strongly connected components on a stack. Both use Tarjan's algorithm to detect components and the time complexity of both is dependent on the time it takes to find the transitive closure of the condensed graph. The worst-case execution time of these algorithms is typically $O(ne + n^2)$.

Nuutila [34] proposes a stack-based approach to generate transitive closure using strongly connected components. An improvement MEMTC is suggested in [24] demonstrating a low memory access map by storing partially handled vertices into a control stack and using a single work stack for adjacent components and vertices. However, due to the inherently sequential depth-first search approach of detecting strongly connected components in sparse graphs, we do not consider parallelizing these approaches. Moreover, these methods are efficiently implemented by learning about the graph structure and properties prior to designing the algorithm. We aim to design a method for general sparse graphs.

## 2.4 Transitive Closure for Sparse Graphs

In real-world graphs, the number of vertices and edges can vary widely with respect to the network being modelled. However, it has been indicated through analysis of numerous networks and pointed out by Barabasi in [8] that most real networks are sparse in nature. This implies that the adjacency matrix of a sparse graph will also be sparse resulting a huge chunk of memory being blocked out with redundant information. A large amount of time will also be spent in redundant operations. In this thesis, we focus on accelerating transitive closure mainly for large scale sparse graphs. Therefore, in the interest of accurately modelling real-world networks and designing an algorithm that is optimized to efficiently analyze such networks, we consider only large scale sparse graphs.

Looking into works related to implementing transitive closure using parallel programming paradigms on GPUs, we explore approaches including sparse matrix multiplication, linear algebra based graph algorithmic implementation, and others. Penn [38] evaluates the performance of various algorithms implemented by matrix multiplication of sparse upper-triangular matrices over closed Boolean semirings using the ZCQ structure. Although it outperforms $n\times$ Dijkstra and Warshall's algorithms for large-scale sparse graphs, the upper-triangular matrix only allows acyclic graphs. We do not consider this method due to requiring efficiency for general graphs. Johnson's [29] algorithm is another extension from $n\times$ Dijkstra for sparse graphs providing better asymptotic performance than Warshall's algorithm. Although, algorithms based on matrix multiplication provide the advantages of simpler implementation and efficiency for sequential approaches, memory management and scalability become drawbacks when implementation needs to be done in parallel. In an attempt to implement All-Pairs Shortest Paths on GPUs, Katz [30] proposed block-based processing of matrices to implement Warshall's transitive closure algorithm. However, this method is inefficient for sparse matrices even when executed in parallel.

GraphBLAS [32] is an API specification that defines a set of sparse matrix operations on an extended algebra of semirings using a range of operators and types. SuiteSparse:GraphBLAS [13] is a complete implementation of the GraphBLAS standard. Graph computations are carried out by applying linear algebraic operations on sparse adjacency matrices. Mutliple implementations, sequential and parallel, have been developed such as the GraphBLAS Template Library [49] we use as a sequential benchmark.

Approaches discussed in earlier subsections based on strongly-connected components depend highly on traversal of graphs [26]. One such intuitive approach is an $n\times$ Depth-First Search that can be referred to as $n\times$ DFS. It requires conducting a repetitive Depth-First Search with the source node iterating over all vertices in the graph. With each DFS traversal starting from vertex $v_1$, every discovered vertex $v_x$ forms a transitive edge, $(v_1, v_x)$ that is added to the transitive closure of the graph. Illustrated in Algorithm 3, the time complexity of this algorithm would be $O(|V| \times (|V| + |E|))$ attributing to the $O(|V| + |E|)$ complexity of a single DFS traversal. However, this approach would require us to compromise either on memory footprint or the degree of parallelism we can achieve. Since our approach is GPU-based, we aim to design an approach that can leverage fine-grained parallelism for higher throughput. Another approach for process-level parallelism using MPI for transitive closure using relational algebra is implemented by Gilray *et al.* in [19]. However, we consider this approach as a benchmark for one of our sequential implementations due to its time and space complexity when implemented as a single process.

A semi-naive evaluation of a relational database query is proposed in [7]. It can be interpreted as an iterative transitive closure algorithm that avoids rediscovery of existing edges in the network, thereby eliminating a majority of the redundant calculations. Our Hash-Based transitive closure approach is derived from the idea

that generating duplicate edges is avoided without the need to maintain a separate data structure for newly generated edges. The motivation for our algorithm also stems from a triangle counting approach. A detailed explanation is provided in Section 1 of Chapter 4.

**Algorithm 2** Warren's Algorithm

```
1: procedure WARREN(A[1...n][1...n])
2:     for i ← 2, n do
3:         for k ← 1, i − 1 do
4:             if A[i, k] then
5:                 for j ← 1, n do
6:                     A[i, j] ← A[i, j] OR A[k, j]
7:                 end for
8:             end if
9:         end for
10:    end for
11:    for i ← 1, n − 1 do
12:        for k ← i + 1, n do
13:            if A[i, k] then
14:                for j ← 1, n do
15:                    A[i, j] ← A[i, j] OR A[k, j]
16:                end for
17:            end if
18:        end for
19:    end for
20: end procedure
```

**Algorithm 3** $n\times$ Depth-First Search Algorithm

**procedure** N-DFS($G(V), G(E)$)
    **for** $u \in G(V)$ **do**
        $DFS(G(V), G(E), u)$
    **end for**
    **return** $R_n$
**end procedure**

**procedure** DFS($G(V), G(E), u$)
    $u \leftarrow VISITED$
    $adj(u) \leftarrow G(E)[u]$
    **for all** $v \in adj(u)$ **do**
        $G(E).append((u, v))$
    **end for**
**end procedure**

# CHAPTER 3

# HORNET GRAPH DATA STRUCTURE

## 3.1 The Hornet Data Structure

The Hornet Data structure is an advanced and complex graph data structure designed to fully support dynamic graph algorithms. This can be accomplished by providing efficiency in the basic graph operations of insertion and deletion of nodes or edges and searching for a node in the graph and by allowing to perform these graph operations in a dynamic manner.

Hornet is built on three components. Block arrays, which are arrays of equally sized memory chunks, are used a particular number of adjacency lists depending on their size. A vectorized bit tree data structure is used to locate empty spaces in block arrays for edge insertion, manage allocation of block arrays until older ones are full to reduce memory footprint and avoid communication overhead. The number of empty spaces in each block arrays are mapped to the address of the corresponding block array using a $B^{+}Tree$ [28]. This makes Hornet powerful in faster updates to the graph for both insertion and deletion.

## 3.2 Principal Advantages of Hornet Relating to Transitive Closure

Hornet is a graph data structure specifically designed for dynamic sparse graph structures. Transitive closure involves finding new edges and inserting them into the graph to explore the graph further for new edges. This requires repetitive and extensive addition of new edges to the graph for each iteration in any type of algorithm. Despite reducing redundancy in exploring new edges, the edge insertion work remains the same for all algorithms. Another challenge is maintaining the edges in the graph to be unique despite repeated attempts to insert duplicate edges.

Apart from providing dynamic updates to the graph, Hornet facilitates bulk edge insertions to the graph with the help of batch updates [15] supported by removal of cross duplicates. This is especially useful for transitive closure due to finding the same transitive edge repeatedly. Avoiding duplicate edge or node insertion is mandatory to avoid errors in estimation of the size of transitive closure and other metrics to analyze performance and results of the algorithm. Moreover, due to Hornet supporting large scale batch updates, it is not required to re-initialize the graph for every iteration. This reduces both memory footprint and time between two iterations of finding new edges significantly. Due to the above mentioned advantages, Hornet provides for an implementation of a transitive closure algorithm, it is perceived as the most appropriate choice of data structure for the project. A high degree of parallelism makes it suitable for large scale graphs as well. In-built operators with load balancing enabled for traversing all nodes and edges in the graph are provided in the Hornet GPU implementation, making it convenient to program and focus on the implementation of the algorithm. More details about it's advantages to our Anti-Section Transitive Closure method are provided in Chapter 4.

# CHAPTER 4

# PARALLEL ANTI-SECTION APPROACH FOR TRANSITIVE CLOSURE

In this chapter, we study the relation of triangle counting to the Anti-Section approach, establish a strong understanding of the approach itself, study the Dual-Round and Hash-Based implementations of the Anti-Section transitive closure algorithm and estimate their time and space complexity.

## 4.1 Identification of New Edges Using Anti-Section

### 4.1.1 Triangle Counting

Given a graph, $G(V, E)$, triangle counting entails counting the number of triplets in $V$ that form a triangle. A triangle is a set of three vertices in $V$ such that they are mutually adjacent in G. Consider the nodes $v_1, v_2 and v_4$ depicted in the graph G1 in 4.1. There exists an edge between all nodes in the triangle. In G1, these three nodes form a triangle.

An intuitive approach to counting triangles in a graph is a brute-force approach that involves enumerating over triplets consisting of distinct vertices and identifying valid triangles. However, there are $(n^3)$ distinct vertex triplets in a graph with number of vertices equal to $n$. Therefore, it becomes a $\Theta(n^3)$ solution with respect to time complexity.

A more efficient approach is to view triangle counting as a set intersection problem. An edge, $(v_1, v_2)$ can form triangles using nodes that connected to both $v_1$ and $v_2$. This develops intuition for an approach that involves iterating over all edges, say $(u, v)$, and intersecting over adjacency lists of $u$ and $v$. Nodes that are common between the adjacency lists of $u$ and $v$ will form a triangle $u$ and $v$. Intersection of two adjacency lists has been approached by binary search, hash maps and sorted set

intersections. Green *et al.* [23] proposed a merge-path based approach supported by an efficient way to partition the two sorted lists. To allow GPU threads to perform the intersection in parallel, the lists are partitioned into balanced and disjoint sub-ranges of the original lists. There are successful works on GPU, such as the list intersection method[17, 21] on GPU that has become an essential kernel algorithm of the Hornet[11] package that can supports dynamic graph calculations. Other methods and implementations of triangle counting using list intersections are proposed in [25, 17, 20, 37].

While the above approaches are focused on finding intersections between adjacency lists of two nodes forming an edge, the Anti-Section approach focuses on finding open wedges. A node that exists in the intersection of two adjacency lists will not be present in the result of Anti-Section on two adjacency lists. The concept is discussed in additional detail in the next subsection.

### 4.1.2 Anti-Section Definition

The Anti-Section of two adjacency lists is an operation to find the non-common elements between them in such a manner that the elements only exist in a particular set of the two. Consider a directed edge between two nodes, $u$ and $v$, denoted as $(u, v)$ and the lists of their adjacent nodes as *adj(u)* and *adj(v)*. The Anti-Section of *adj(u)* and *adj(v)* is defined as the elements present in *adj(v)* but not in *adj(u)*. More formally Anti-Section of *adj(u)* and *adj(v)* can be defined as follows,

$$\begin{aligned} antisection(adj(u), adj(v)) &= \overline{adj(u)} \cap adj(v) \\ &= adj(v) - adj(u) \end{aligned}$$

where $\overline{adj(u)}$ is the complement of the set of vertices in adjacency list of node u. The elements in the resultant of the Anti-Section must belong to *adj(v)* but should

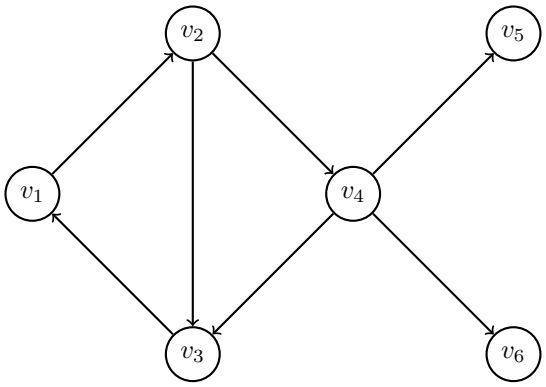



**Figure 4.1** An example graph, G1.

not belong to *adj(u)*. We know that, there is a path from node u to every node that belongs to *adj(v)*. However, the Anti-Section operator helps us eliminate the nodes that do not propagate the paths after $(u, v)$ further. If a node p is present in the Anti-Section of *adj(u)* and *adj(v)*, it implies that there is a path from $u \rightarrow v \rightarrow p$ but there is no edge *(u,p)* in the graph. The edge *(u,p)* is added to the transitive closure of the graph. The concept also relates to finding the set difference of two lists.

Consider the graph G1 in 4.1. For edge $(v_2, v_4)$, the Anti-Section is obtained as follows.

$$adj(v_2) = [v_3, v_4]$$

$$adj(v_4) = [v_3, v_5, v_6]$$

$$antisection(adj(v_2), adj(v_4)) = [v_5, v_6]$$

The new edges added to the transitive closure of G1 will be $(v_2, v_5)$ and $(v_2, v_6)$. Node $v_3$ is in the intersection of the two adjacency lists and hence, would form a triangle. We use operation to find new edges that must be inserted in the transitive closure for each edge in the input network. The algorithm used to generate an Anti-Section for a pair of vertices in the graph requires iterating through both adjacency lists depicted in Algorithm 5. Given that adjacency lists of both vertices are sorted, we skip all vertices in the list of the source node U that are lesser than the vertices

**Algorithm 4** Anti-Section Algorithm

```
1: procedure ANTI-SECTION(U, V)
2:     u_i ← 0, v_i ← 0, count = 0
3:     SRC ← u
4:     ANTISECTION ← ∅
5:     while v_i < |V| do
6:         while U[u_i] < V[v_i] AND u_i < |U| do
7:             u_i ← u_i + 1
8:         end while
9:         if U[u_i] == V[v_i] then
10:            v_1 = v_1 + 1
11:        end if
12:        if V[v_i] ≠ SRC then
13:            count ← count + 1
14:            ANTISECTION.add(SRC, V[v_i])
15:            v_1 = v_1 + 1
16:        end if
17:    end while
18: end procedure
```

in the list of the destination node V. A new edge, $(SRC, v_i)$ is added when we find an element in V that does not occur in U. Until all elements in V re checked, we keep repeating this process. Whenever a common element is found in the two lists, it is ignored by incrementing the pointer. Hence, the name Anti-Section. For every iteration of transitive closure, this operation is applied on all edges in the graph. Further details regarding the usage and implementation are provided in Section 2 of this chapter.

## 4.2 Iteration based Transitive Closure Implementation

In Section 2.3, we discussed three approaches proposed to generate transitive closure. Direct transitive closure algorithms rely on a carefully chosen processing order rather iteration for termination[2]. Direct algorithms are often efficient and easy to implement in a sequential implementation. However, the carefully chosen order can often be harmful for massive parallel processing so we choose iterative method to implement our algorithm.

The general idea behind transitive closure is based on iterative traversal of the input network to discover new transitive edges in each iteration that can be added to the input network to obtain the result for the subsequent iteration. Pseudo-code for our iteration based approach is provided in Algorithm 5. A batch is maintained for newly discovered edges all over the graph. In each iteration, the Anti-Section operation is run for all edges in present in the graph and newly discovered edges are added to the global batch. The global batch is then inserted in the graph after removal of duplicates once Anti-Section operations over all edges in the graph are completed. The graph is then sorted in order to prepare it for the next iteration.

Since the Anti-Section operation is run for a different directed edges resulting in more directed edges to be inserted in the graph, it is important that edges should not be inserted immediately as they are found. If an edge is inserted into the main

**Algorithm 5** Iteration-based Transitive Closure Algorithm

```
1: procedure ITERATION-BASED TRANSITIVE CLOSURE(G(V), G(E))
2:     repeat
3:         E ← G(E)
4:         globalBatch ← ∅
5:         for all (u, v) ∈ E do in parallel do
6:             threadBatch ← ANTISECTION(adj(u), adj(v))
7:             if threadBatch ≠ ∅ then
8:                 globalBatch.update(threadBatch)
9:             end if
10:        end for
11:        G(E).INSERT(globalBatch)
12:        G(E).SORT()
13:    until globalBatch ≠ ∅
14: end procedure
```

graph as soon as it is found, it is possible that it is inserted in an incorrect position impacting Anti-Sections negatively. To keep track of all adjacency list that are in modification at a given time will become complex due to the number of edges and the number of threads running at a single point. During every iteration, the newly discovered edges must be inserted in the graph such that they can be accessed in a sorted order in the next iteration. These newly inserted edges are considered in subsequent iterations to propagate the transitive closure result. Efficiently inserting edges in the graph such that it does not affect traversal in future iterations is a concern for a parallel implementation. It also raises concerns over repeated attempts to insert duplicate edges to the graph increasing time complexity due to the time it takes to locate whether a certain edge is present in the graph. Although the lookup time for

an edge might be very low, repeated lookup scaling to the number of transitive edges contributes significantly to the running time of the algorithm.

Our implementation inserts edges in the bulk synchronous fashion. During the iteration when new edges are being discovered, they are recorded and stored in an auxiliary list of edges. Due to the process of discovering new edges being parallel, the addition of edges to the graph is done in bulk after all new edges for that iteration are discovered.

Another advantage of bulk insertion is that is gives us the liberty to ensure that duplicate edges are not inserted in the graph. It is possible that the edges sets resulting from the Anti-Section of two pairs of vertices where the destination vertex is different has a non-empty intersection. For example, in 4.1, consider the Anti-Section sets for edges $(v_2, v_3)$ and $(v_2, v_4)$. Vertex $v_6$ will appear in both, the Anti-Section of $(adj(v_2), adj(v_3))$ and $(adj(v_2), adj(v_4))$. This will result in the the discovery of transitive edges $(v_2, v_6)$ twice. Bulk synchronous insertion allows us to eliminate duplicate edges like this one and insert only unique edges to reduce edge insertion overhead.

The only remaining concern is the generation of the global batch containing edges that are to be inserted in the graph. Since we focus on large scale graphs, there may be a large number of new edges discovered and a lot of the edges might be duplicates in the batch itself. Detection of duplicates after an attempt to insert in the graph is an expensive operation. We discuss a two-pass Anti-Section approach and a Hash-Based Anti-Section approach in the subsequent subsections to understand the creation of the bulk edge set.

### 4.2.1 Dual-Round Anti-Section Transitive Closure

The Dual-Round implementation of transitive closure includes two passes of Anti-Sections over the graph during each iteration. The first pass counts the number of

edges discovered in that iteration. Then an array of size equal to the exact number of edges is allocated. In the second pass, edges are inserted in this allocated array as they are discovered by Anti-Sections. The operation for the second pass during an Anti-Section follows Algorithm 4 barring the operation on Line 13. It adds new edges to the "ANTISECTION" set which is analogous to the array we allocate after the first pass. For the second pass, only the counting operation on Line 13 is conducted and the adding operation in skipped.

Due to the size and sparsity of the graph, it is highly probable that new edges discovered in the same iteration are duplicates. Therefore, the array containing the new edge set might consist of duplicate edges. Insertion of duplicate edges is handled by Hornet. However, the batch size of the bulk edge set to be inserted can be reduced significantly by removing duplicates reducing the time taken to insert the batch of edges for each iteration. The time complexity of this implementation is not affected by running two passes for each iteration. However, the running time is increased.

Moreover, more time is spent between the two passes to weed out duplicate edges and insert edges unique to the new edge set. Although Hornet can handle the situation when duplicate edges are inserted, the internal storage structure uses block arrays to store edges. The adjacency list of one vertex can span over multiple block arrays. Although, pointers are available for the memory location of each vertex in the adjacency list, it increases time due to memory access. Therefore, filtering duplicates before insertion is an attempt to reduce the amount of work done to eliminate duplicates in the final graph. More details about the complexity bound of this algorithm are discussed in subsection 4 of this section.

### 4.2.2 Hash-Based Anti-Section Transitive Closure

This approach is specifically designed to reduce the run time with respect to running two passes of Anti-Sections and filtering duplicate edges in every phase. The

implementation of Dual-Round approach indicated a high number of duplicates for each iteration. Therefore, a lot of the memory allocated based on the count in the first pass was allocated unnecessarily. Moreover, the number of duplicates increased with increase in the number of iterations. This resulted in requirements for large amounts of memory ultimately slowing down implementation due to duplicate removal. The Hash-Based approach is designed to reduce time consumed due to duplication.

The hash table is implemented as a simple array containing '0' or '1' based on whether an entry is exists or not. The process after an edge $e = (u, v)$ is discovered occurs in the following manner and as shown in Operations 14 to 17 in Algorithm 6.

1. Consider that the hash function $h$ is used to map an entry to our $HT\_Array$.

2. With the edge, e, we calculate the hash of e, $h(e)$ to obtain it's index in our hash table.

3. We check the value at $HT\_Array[h(e)]$. If it is equal to '0', we update it to '1' indicating that the respective edge should be added to the batch. If the value is already '1', either the edge has been added to the table or a different edge has been added and has the same hash value as this edge. We do not attempt to insert this edge as there is no way to know which of the two has occurred.

Considering the above approach, there are concerns over the scenario where two edges have a colliding hash value and are also discovered in the same iteration. It introduces doubt about the possibility of skipping edges in that iteration and missing from the final transitive closure. However, due to the iteration-based approach of our implementation having an exit condition of an empty set of newly discovered edges, this will not happen. Consider that we discover two edges, $e$ and $f$ with the same hash values in the same iteration. If $e$ is inserted first in the hash table, $f$ will not be inserted. The hash table is then cleared for the next iteration. After $e$ is inserted in the graph, a subsequent iteration will find $f$. The transitive closure will

**Algorithm 6** Hash-Based Anti-Section Algorithm

1: **procedure** ANTI-SECTION($U$, $V$, $HT\_Array$)
2: $u_i \leftarrow 0$, $v_i \leftarrow 0$, count $= 0$
3: $SRC \leftarrow u$
4: $ANTISECTION \leftarrow \emptyset$
5: **while** $v_i < |V|$ **do**
6: **while** $U[u_i] < V[v_i]$ AND $u_i < |U|$ **do**
7: $u_i \leftarrow u_i + 1$
8: **end while**
9: **if** $U[u_i] == V[v_i]$ **then**
10: $v_1 = v_1 + 1$
11: **end if**
12: **if** $V[v_i] \neq SRC$ **then**
13: $count \leftarrow count + 1$
14: $hashVal = Hash(SRC, V[v_i])$
15: **if** $HT\_Array[hashVal] == 0$ **then**
16: $ANTISECTION.add(SRC, V[v_i])$
17: $HT\_Array[hashVal] \leftarrow 1$
18: **end if**
19: $v_i \leftarrow v_i + 1$
20: **end if**
21: **end while**
22: **end procedure**

only terminate if zero new edges are found independent of the edges present in the graph.

The size H of the hash-table is chosen arbitrarily. Adjusting the size of the hash-table according to graph size and structure can be achieved to reduce number of iterations. There might be concerns over missing a few edges given the hash-table size might be bigger than transitive closure of the graph. However, these concerns are addressed earlier due to the logic of the terminating condition being that no new edges are found in that iteration. It has been found that the larger the hash-table size, the lesser the run time and higher the space occupied.

**Hash Function:** For calculating the hash of an edge, we use the MurmurHash3 hash function [12] designed by Austin Appleby. The *indexvalue* is calculated as $hash(e) moduloH$. The lookup in $HT\_Array$ is done using this index value as $HT\_Array[index]$.

### 4.2.3 Time and Space Complexity Analysis

**Trade-offs for Dual-Round and Hash-Based approaches:** The Hash-Based approach saves on run time by eliminating duplicate edges and reduces the overall memory footprint by avoiding the allocation of the array. It also helps us avoid scanning the entire graph twice. However, the space complexity of storing a hash-table of size H is $O(H)$ which is might be more than the array allocated for the Dual-Round approach. An appropriate size for the hash-table must achieve balance between the number of iterations and the amount of memory allocated. A hash-table of significant size can reduce the number of iterations but result in a larger memory footprint. A small hash-table results in a high number of collisions for newly discovered edges which are then postponed to be inserted in further iterations. This results in a smaller memory footprint but a larger number of iterations. Therefore, with respect to time complexity, a tight bound cannot be given for the number of iterations that

the algorithm will require for completion. What the Hash-Based approach loses in complexity of estimation of space, it compensates in reducing the number of Anti-Section operations from that in Dual-Round to a half and also does not require removal of duplicates. The Hash-Based approach performs better than the Dual-Round Approach.

**Analysis of Time Complexity:** First we analyze the time complexity of a single Anti-Section operation conducted on an edge $e = (u, v)$. Let us denote the degree of the vertices, $u$ and $v$ as $d_u$ and $d_v$ respectively. It is equivalent to the length of their respective adjacency lists. Considering that our Anti-Section operation scans through the adjacency list of the destination vertex in the edge $v$ while iterating through the adjacency list of the source vertex $u$, we require $O(d_u + d_v)$ operations for a single Anti-Section. The upper bound for processing one vertex of the graph, thus becomes $O(max(d_u, d_v))$.

Let us also consider that with each iteration, the value of $d_u$ for all $u$ in the graph might be changing. Therefore, the complexity of processing one vertex for the $i^{th}$ iteration is $O(d_u^i)$.

Since vertices may have varying out-degrees affecting the length of their adjacency lists, we limit the Big O complexity using the degree of the vertex having the highest out-degree $d_{max}$ in the graph. Considering an edge with maximal degree, $d_{max}$, the upper bound for all edges in the graph is also $d_{max}$ making the time complexity of processing one edges for the $i^{th}$ iteration $O(d_{max}^i)$.

Therefore, iterating over all the edges per iteration results in a time complexity of $O(|E| \cdot d_{max}^i)$. Since, the upper bound for the number of edges can be estimated as $O(d_{max} \times |V|)$, we re-write the time complexity of an iteration of Anti-Section as $O(|V|.{d_{max}^i}^2)$.

Now, we derive the number of iterations and the ending condition to estimate the time complexity of the algorithm as a whole.

The ending condition is specified in Operation 13 of Algorithm 5. The algorithm stops iterating over edges when there are no new edges to be added to the edge set. This means that the longest possible path is explored in the transitive closure is the diameter of the graph formed after transitive closure.

When we apply an Anti-Section operation on a transitive closured edge, it results in more transitive edges that are twice as long as the paths connecting this edge. For example, if we have an edge $e = (u, v)$ with no intermediate vertices and we obtain another edge $e_1 = (u, w)$ when we perform an Anti-Section operation on $e$, it means that $v$ is an intermediate vertex in the transitive edge $e_1$. In the first iteration, the longest path added as an edge is of length 2. Further, when an Anti-Section operation is applied on $e_1$, more edges such as $(u, x)$ will be added, where $x$ is a neighbor vertex of $w$ such that it is not a neighbor of $u$. In this case, the path between vertices $u$ and $x$ looks like $(u, v, w, x)$. In the second iteration, the length of the longest transitive edge is 4. Further, with every iteration this length is doubled due to the Anti-Section operation being applied on transitive edges of an arbitrary length. Since the longest possible path in a graph is $|V| - 1$, the number of iterations it takes for our algorithm to discover this edge is $\lceil log_2|V| \rceil$. This gives us an upper bound for the number of iterations in our algorithm.

We now move forward to estimating the time complexity of both algorithms. Fundamentally, the only different operations in both algorithms are those of constant time. Hash lookup and insertion into the new edge set takes constant time for the Hash-Based approach. For the Dual-Round approach, it takes an equivalent amount of operations for incrementing counter and insertion into new edge set. Although, the Anti-Section operation remains of the same time complexity, operations between iterations differ in complexity for both the algorithms. The batch edge insertion and graph sorting operations remain the same. Each adjacency list is sorted separately using a segmented-sort algorithm presented by Green *et al.* [18]. Sorting an adjacency

array of size $d$ has a time complexity of $d \cdot log(d)$ or $O(d)$ depending on the sorting algorithm used (merge-sort and radix-sort, respectively, in our analysis). When compared with the time complexity of the Anti-Section operation requiring $O(d^2)$ operations, the overhead of the sort is not high.

For the Hash-Based approach, we reset the hash-table between iterations along with the new edge set. For the Dual-Round approach, we filter the new edge set between iterations. The size of the new edge set is proportional to the number of Anti-Section operations. Therefore, the time complexity for both approaches is bound at

$$O(\sum_{i=0}^{log_2(|V|)} |V| \cdot (d^i_{max})^2) \tag{4.1}$$

where i stands for the iteration number.

**Analysis of Space Complexity:** We look into the memory footprint of the algorithm due factors other than the space required to store the graph using Hornet. For both approaches, an array is allocated for storing newly discovered edges at each iteration. Since we add the new edges to the graph itself and do not initialize another intermediary graph, we rely on estimating the space complexity of Dual-Round through the size of the new edge set and that of the Hash-Based approach with respect to the size of the hash-table. Considering that Hornet requires $O(|E| + 2|V|)$ space for a graph, the space occupied by the resultant graph is $O(|E|^i + 2|V|)$ for the $i^{th}$ iteration. The Dual-Round approach allocates an array of size equal to the number of edges discovered. Considering the maximum out-degree value for each vertex, the size of the edge set array can be bound at $O(|V| \cdot d^i_{max})$. Therefore, the space complexity of the Dual-Round approach can be bound at $O(|V| \cdot d^i_{max} + |E|^i + 2|V|)$ and can be re-written as

$$O(|V| \cdot d^i_{max}) \tag{4.2}$$

where $i$ is the iteration number.

For the Hash-Based approach, the size of the new edge set is derived in a similar manner. Since there is equal possibility of finding unique edges, the preliminary evaluation remains the same. However, we need to consider the size $H$ of $HT_Array$. Therefore, the space complexity of the Hash-Based approach can be bound at

$$O(|V| \cdot d^i_{max} + H) \tag{4.3}$$

where $H$ is the size of the hash-table and $i$ is the iteration number.

# CHAPTER 5

# EXPERIMENTAL SETUP

In this chapter, we describe the hardware and software environment used to implement our algorithm. We further describe the input graphs we used for our experiments. Then, we go into additional detail about the sequential and parallel TC implementations we compare our algorithm against and the frameworks used to implement them.

## 5.1 Hardware

**GPU Implementation:** The Anti-Section Transitive Closure algorithm is implemented to be suitable for large scale sparse graphs. The new generation of GPUs are fundamentally multi-threaded stream processors. A number of applications that are traditionally run on the CPU have started being implemented again to run on the GPU. This technique is called general-purpose computing on graphics processing units (GPGPU). NVIDIA offers programming interfaces for making GPGPU accessible to programmers who do not possess a deeper understanding of programming for computer graphics.

NVIDIA's Compute Unified Device Architecture (CUDA) offers a higher level C-like API. The CUDA platform is unified in the sense that it has no architectural division for vertex and pixel processing. They offer band-width and single-precision floating point arithmetic computation rates in high amounts. Stream processing is basically when a single data parallel function (kernel) is executed on a stream of data. The CUDA programming model is based on this style of processing. A CUDA program is composed of two parts, a device, the GPU in our case, that implements the kernel, and a host or the CPU which makes calls to the kernel based on the degree of parallelism specified. We harness the power of fine grained

parallelism by implementing the Anti-Section operation for a single iteration in parallel. The streaming multiprocessors (SMs), sometimes called the GPU chips are the fundamental building blocks of recent NVIDIA GPUs. A kernel is executed by many threads on the GPU. These threads are organized as a grid of thread blocks and are also known as streaming processors. The grids are batches of threads that can coordinate through on-chip shared memory and synchronize their execution. Each thread block is executed by only one SM, but each SM can execute multiple thread blocks simultaneously.

Our algorithms are implemented in CUDA using NVIDIA GPUs on the Lochness GPU cluster at the New Jersey Institute of Technology. The GPU used for this experiment is the NVIDIA Titan V. The TITAN V is built on the 12 nm process, and based on the GV100 graphics processor. The GV100 graphics processor is a large chip with a die area of 815 $mm^2$ and 21,100 million transistors. It features 5120 shading units, 320 texture mapping units, and 96 ROPs. NVIDIA has paired 12 GB HBM2 memory with the TITAN V, which are connected using a 3072-bit memory interface. The GPU is designed based on the Volta micro-architecture and can run 5120 CUDA threads due to the 80 cores (SMs) and 64 CUDA threads. The CUDA version used is 10.2.

**CPU Implementation:** The implementation of sequential approaches was conducted on an Intel Xeon E5-2650 v4 Processor running 2.2GHz with 30MB LLC with 512 GB memory in total.

## 5.2 Input Network Datasets

All input networks for our tests are taken from SuiteSparse. The SuiteSparse Matrix Collection, formerly known as the University of Florida Sparse Matrix Collection, is a large set of sparse matrices that arise in real applications.

**Table 5.1.** Input Networks Used for Experiments

| **Graph Name** | $\|V\|$ | $\|E\|$ | $\|\hat{E}\|$ [a] | Sparsity of TC |
|---|---|---|---|---|
| p2p-Gnutella09 | 8,114 | 26,013 | 21,400,336 | 0.6745 |
| delaunay_n13 | 8,192 | 24,547 | 1,656,270 | 0.75 |
| as-22july06 | 22,963 | 48,436 | 1,477,572 | 0.971 |
| delaunay_n15 | 32,768 | 98,274 | 4,703,128 | 0.956 |
| delaunay_n17 | 131,072 | 393,176 | 23,499,929 | 0.9986 |
| cit-HepPh | 34,546 | 421,578 | 485,646,072 | 0.5931 |

[a] $|\hat{E}|$ is the number of edges in the transitive closure of respective graphs.

The transitive closure of an undirected real-world graph typically consists of the entire graph. For an undirected graph, when a connected component is identified, it means that any two nodes in the connected component will have an edge between them in the transitive closure. This means the transitive closure of an undirected graph will result in a large number of nodes connected to each other also known as a clique. Storing such an instance will require a substantial amount of memory. Therefore, we select a variety of real-world and synthetic matrices from SuiteSparse.

The properties of each network and their transitive closure is reported in 5.1. *p2p-Gnutella09* is a peer-to-peer file sharing network from August 2002. The *delaunay_n13, delaunay_n15,* and *delaunay_n17* are selected from a group of synthetic graphs that have been generated as Delaunay triangulations of random points in the unit square. *as-22July06* is a symmetrized snapshot of the structure of the Internet at the level of autonomous systems. *cit-HepPh* is a citation network of papers in the high energy physics phenomenology section of arXiv. It is interesting to note that the transitive closures of the most of the graphs maintain their sparsity like the original graph.

## 5.3 Sequential Implementations

In this section, we look at two sequential implementations of the Transitive Closure algorithm. One is NetworkX based implementation and the other is based on GraphBLAS (SEI-GBTL).

### 5.3.1 NetworkX Implementation

NetworkX is a Python based network analysis framework that provides tools for studying the structure and dynamics of networks with a standard programming interface and graph implementation.

We implement transitive closure on the NetworkX directed graph structure using an in-built API for obtaining transitive closure. The underlying implementation uses an iterative Depth-First Search algorithm. A depth-first search is initiated from every node in the graph as source and each node visited is used to find edges to add to the transitive closure with each iteration.

### 5.3.2 GraphBLAS Implementation

GraphBLAS is an API specification that is based on linear algebra to define standard standard building blocks for graph algorithms. It describes formulations of using linear algebraic methods for efficiently defining, traversing and manipulation graphs. Various implementations are contributed by representatives in the GraphBLAS forum to provide a bridge between low-level tuning specific to varying hardware and high-level construction of graph algorithms.

We use one such open-source implementation called the GraphBLAS Template Library, abbreviated as SEI:GBTL. This implementation of GraphBLAS is available for both generic CPU systems and GPU systems. It provides the GraphBLAS API as a $C++$ library with common graph algorithms built on top of it. We utilize the sparse matrix multiplication operator provided in the library with a single thread.

## 5.4 GPU Implementation

The NVIDIA cuSparse library provides a set of basic linear algebra subroutines that are designed for sparse matrices. It can be called using $C$ and $C++$ and is implemented on top of an NVIDIA CUDA runtime. It provides various options for sparse representation of networks.

We implement cuSparse-based Transitive Closure by following the sparse matrix multiplication approach. Multiplying an adjancency matrix of a graph with itself results in a matrix with transitive edges extending one edge further. Multiplying the adjacency matrix with itself $|V|$ times results in the transitive closure of graph. We utilize the *cusparseXcsrgemmNnz* operator to implement sparse matrix multiplication on an NVIDIA GPU using CUDA C++.

# CHAPTER 6

# EXPERIMENTAL RESULTS

## 6.1 Performance Evaluation

We compare the performance of our Anti-Section transitive closure algorithm against those of the high-performance implementations based on linear algebra and sparse matrix multiplication and against that of the NetworkX implementation based on an $n \times DFS$ algorithm. The high-performance implementations include a GraphBLAS based implementation which is implemented sequentially and a cuSparse-based implementation which runs on a GPU. Fig. 6.1 depicts the execution time for each of the implementations. A lower execution time is observed for the Hash-Based and Dual-Round Anti-Section implementation as anticipated. With the exception of the Dual-Round implementation not finishing and being outperformed by cuSparse for the input graph *p2p-Gnutella09*. Although cuSparse does not complete execution for input graphs *delaunay_n15* and *delaunay_n17* due to its size, our implementations outperform the rest. Missing bars in the plots are due to the execution not completing for the respective graph as a result of segmentation faults, memory overflow and process time-outs. We comment on all comparisons considering instances where implementations have completed for the benchmark implementations.

In Fig. 6.2 (a), speedup for the Dual-Round Anti-Section transitive closure algorithm can be observed against both sequential and one parallel implementation. A speedup value on the Y-axis exceeding 1 denotes that the implementation has outperformed the baseline. The Dual-Round algorithm gives over 300× speedup and 200× speedup over the NetworkX and GraphBLAS (SEI-GBTL) implementation, respectively. The deficiency of the Dual-Round implementation against cuSparse for three of the input graphs can be attributed to time taken to remove duplicate

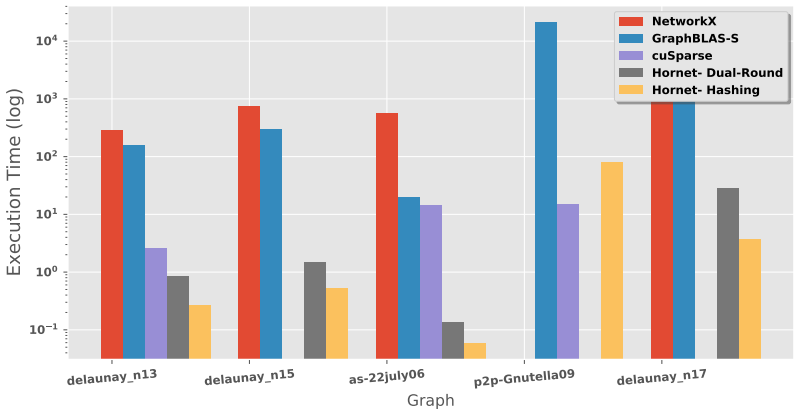
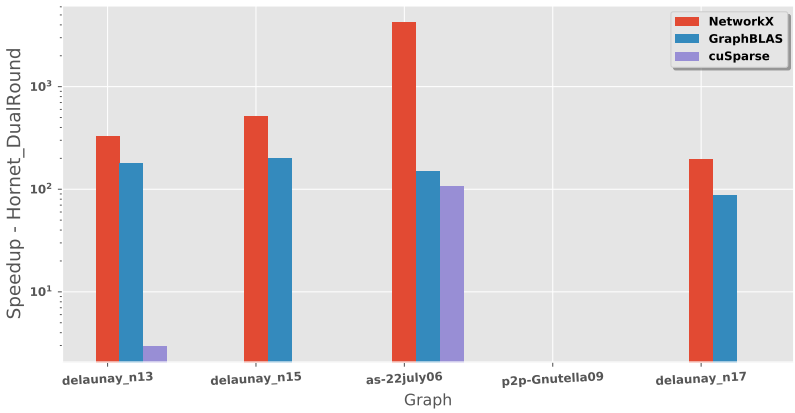



**Figure 6.1** Log-scale plot of execution time taken by various transitive closure implementations.

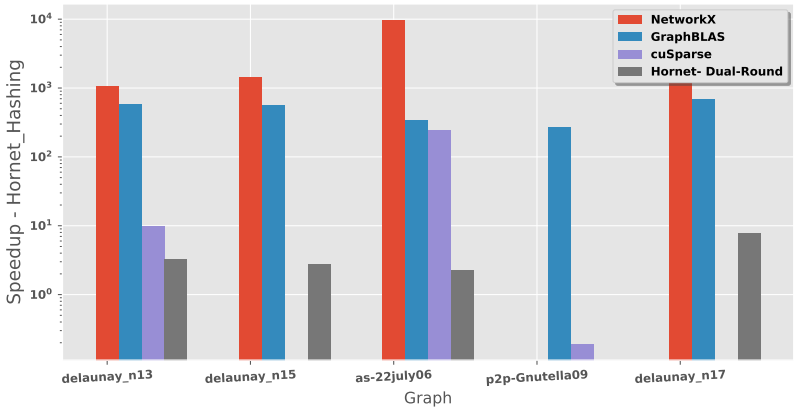
edges when the size of the new edge set is very large. Therefore, even though the Anti-Section phase takes longer due to a double pass, other phases consume more time.

In Fig. 6.2 (b), the speedup for the Hash-Based approach can be observed as our implementation outperforming all implementations for all inputs except cuSparse for *p2p-Gnutella09*. The Hash-Based Anti-Section algorithm is observed to be 3X-5X faster than the Dual-Round algorithm. The Hash-Based algorithm outperforms NetworkX by 1000× for all inputs and outperforms the GraphBLAS SEI-GBTL implementation by 400×. This can be attributed to a bottleneck caused by memory overflow due to large edge sets despite Hash-Based methods eliminating a large number of duplicates. For one relatively smaller graph, cuSparse is about 6× quicker than our algorithm. There is also an instance of the smallest graph where the Hash-Based approach is over 100× faster than cuSparse.

NetworkX is observed to take the highest execution time for all implementations due to the implementation being Python-based. The sequential GraphBLAS implementation outperforms NetworkX for larger graphs due to the edge linear algebra-based implementations provide. Moreover, GraphBLAS is a C++ based




**(a)** Dual-Round Speedup




**(b)** Hash-Based Speedup

**Figure 6.2** Speedup plot for Dual-Round transitive closure (top) and Hash-Based transitive closure implementations using Anti-Section approach(bottom).

implementation. GraphBLAS SEI-GBTL could have been implemented with the use of multiple threads to better analyze the performance of our algorithm against parallel benchmarks.

Both the GraphBLAS SEI-GBTL and cuSparse implementations are built on the same underlying algorithm for generation of transitive closure. Therefore, the parallel GPU implementation of cuSparse are observed to be approximately 50× faster than the single-thread GraphBLAS SEI-GBTL. A parallel CPU implementation of GraphBLAS SEI-GBTL will give a much better performance owing to the number of

threads and cores in the processor. Therefore, the gain provided over GraphBLAS SEI-GBTL by our algorithm is due to the the Anti-Section formulation for discovery of new edges along with the compute advantage provided by the GPU.

## 6.2 Number of Iterations

The performance of our algorithm is largely dependent on the number of iterations required due to the exit condition of not finding any new edges. This is also important to analyze correctness of the algorithm in the context of the number of iterations it takes for it to converge. Therefore, we compare the number of iterations taken to generate the transitive closure of a graph in various implementations in Fig. 6.3. Results for the NetworkX DFS implementation are not included due to the meaning of one iteration being on DFS and hence, resulting in $|V|$ number of iterations. The plot signifies that the numbers of iterations for all implementations are comparable and nearly equal.

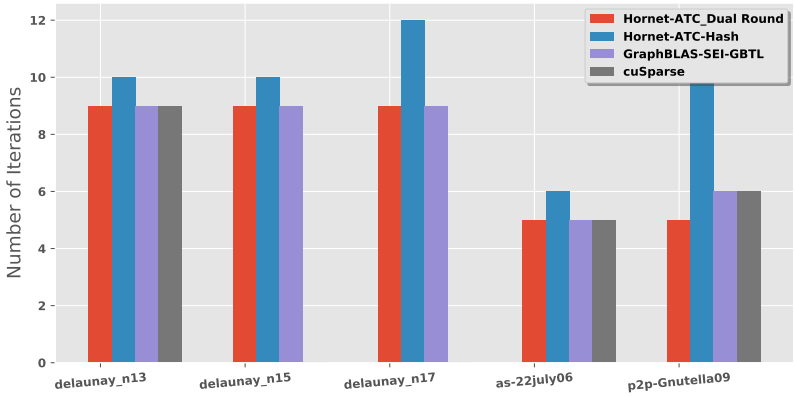



**Figure 6.3** Number of iterations needed for finding the transitive closure for various inputs.

However, the slight increase in the number of iterations for the Hash-Based approach is attributed to the edges remaining undiscovered due to collisions in the hash value of newly discovered edges. On the other hand, the Dual-Round approach does not have any edge filter and hence, performs better in that case by one iteration.

But the Hash-Based approach compensates in the form of each iteration being one single pass as opposed to the double pass in the Dual-Round approach. The number of iterations required will always be greater for the Hash-Based method since the additional iterations are appended due to the false-positive detection of duplicate edges. However, the variance of the number of iterations is not substantial indicating that the Hash-Based approach is more efficient to save time with respect to removing duplicates and doing two passes of Anti-Section operations.

### 6.3 Phase-wise Analysis of Anti-Section Transitive Closure Performance

A breakdown of the phase-wise execution time for both the Dual-Round and Anti-Section algorithms is depicted in Fig. 6.4. Three networks of varying size and sparsity were selected for the analysis. The left column in Fig. 6.4 illustrates the percentage of total execution time taken by each of the phases for both algorithms in each iteration. The phases include the Anti-Section operation itself, bulk edge insertions and sorting the graph.

It must be noted that the time taken by each phase must be viewed as a percentage of the total execution time and that the time taken for phases such as sorting and edge insertion cannot be compared simply based on this percentage. The time taken to sort a graph will be approximately equal for both, the Dual-Round and the Anti-Section approach. However, in the plot, it appears as if sorting takes longer for the Anti-Section approach. However, that is not the case. Since the Anti-Section operation is executed twice for the whole graph in each iteration for the Dual-Round algorithm, it consumes a larger portion of the total time for execution. Therefore, the time taken to sort the graph is significantly lesser in terms of percentage of the total time of Dual-Round. Sorting appears as a major part in the Anti-Section approach due to the Anti-Section operation being executed only once. Moreover, the process of edge filtering must also be taken into consideration for the Dual-Round approach.

The disparity between the Anti-Section phases of both algorithms is more than double due to edge filtering before bulk insertion. This also results in the massive 5× speedup of the Hash-Based approach over the Dual-Round approach.

The time taken for sorting the graph varies according to the size of the graph. Therefore, the time taken for sorting the graph increases as more edges are added with each iteration in the graph. However, the time spent on Anti-Section increases further due to the larger number of edges. Due to the scale of performing Anti-Section operations for each edge in the graph, the relative cost of sorting with respect to time reduces. It can also be attributed to less edge insertion towards the end of execution. It has also been observed that lesser edges are added in the last few iterations.

### 6.4 Performance Analysis With Respect to Size of Hash-table

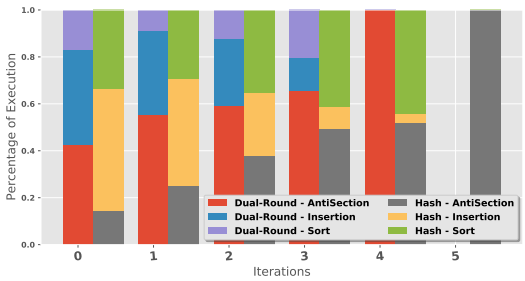
It is important to analyze performance with respect to the size of hash-table due to the relation of number of iterations with the collision rate of edges. The number of iterations increase if edges are detected as existing in the hash table when they are not due to two edges having the same hash value. Therefore, it is assumed that the number of iterations increase with the decrease in the size of the hash table. Fig. 6.5 depicts the influence of the hash-table size on the number of iterations and as a result on the execution time of the Hash-Based transitive closure. It can be seen how additional iterations are caused by a small hash-table. However, a larger hash-table results in a larger memory footprint and ultimately more time to reset the hash-table. However, the hash-table is designed as a simple array and hence, resetting is inexpensive relative to the Anti-Section operation.

Moreover, it can also be observed in 6.5 (b) that the variance of execution time over different hash-table sizes for each graph is not very high. Therefore, the trade-off between memory and time might not be significant. Increasing the size of the hash-table by 100 million results in minor improvement in execution time indicating

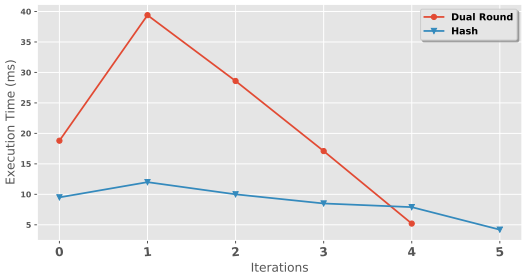
that the number of collisions are relatively lesser. However, this might be an issue for much larger graphs. An optimal value can be chosen by allocating the hash-table ten percent of the system memory. For the worst case of a large transitive closure result, the entire system memory will be full after ten iterations, since unique edges will be added by the hash-table every single. Fewer iterations will be required when the hash-table is not utilized completely for smaller graphs.

### 6.5 Size of New Edge Set Per Iteration

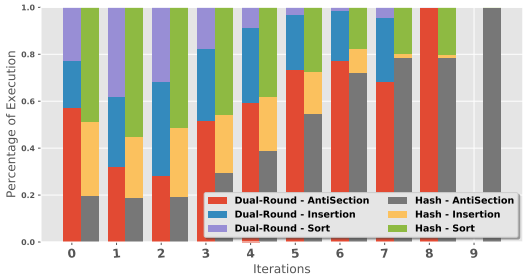
We analyze the execution time of an iteration as a function of the size of the batch of inserted edges for both our Anti-Section algorithms and cuSparse implementation as depicted in Fig. 6.6. The Hash-Based Anti-Section approach is requires approximately $3\times to 5\times$ lesser time than the Dual-Round approach. The iterations for Hash-Based Anti-Section operations are typically observed to be over approximately $10\times$ faster than cuSparse. Some instances are observed such that it is over $100\times$ faster. The exception of the Anti-Section algorithm being slower than cuSparse only holds for several iterations for the *p2p-Gnutella09* input network. This can be attributed to a large number of edges being added to the graph. The time-wise cost of Anti-Section operation increases with increase in density of the graph as a result of more number of edges.



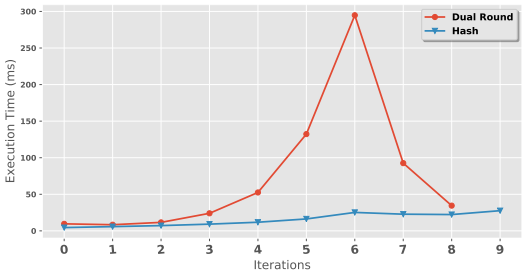

**(a)** as-22July20



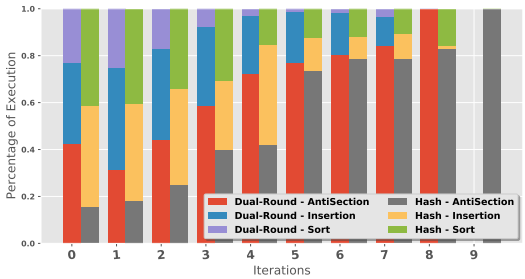

**(b)** as-22July20



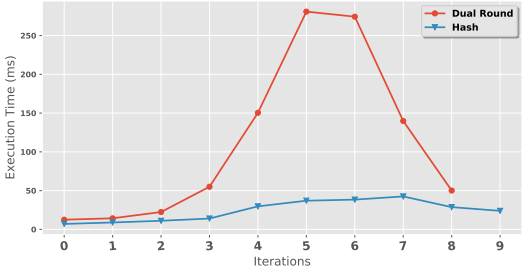

**(c)** delaunay-n13




**(d)** delaunay-n13

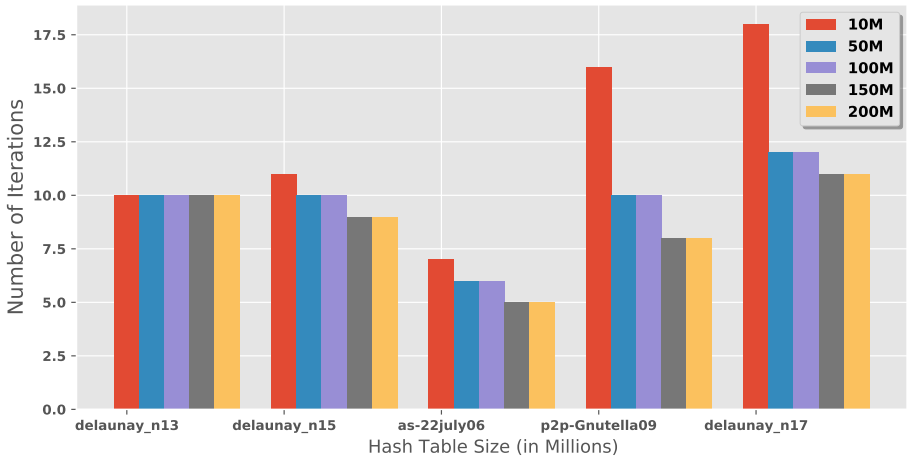



**(e)** delaunay-n15

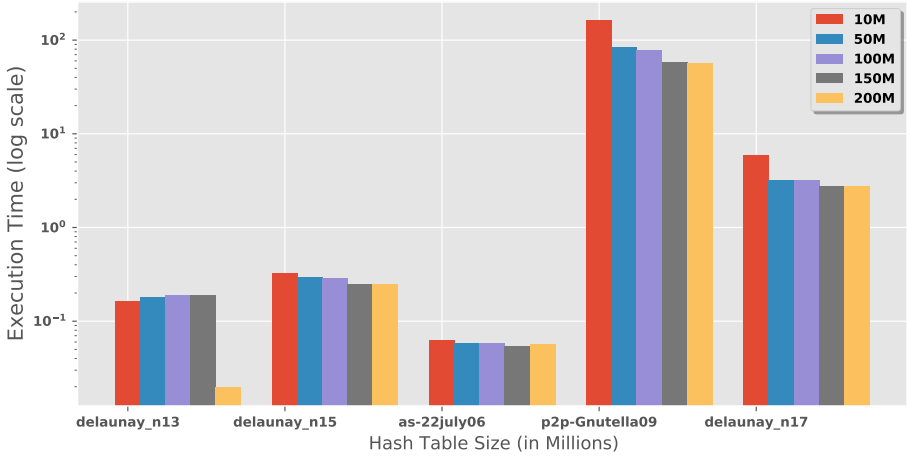



**(f)** delaunay-n15

**Figure 6.4** The left column depicts the execution breakdown (Anti-Section, bulk insertions, and graph sorting) in percentage for three selected networks as a function of the iteration. The execution breakdown is relative to the execution time of a given algorithm. The right column depicts the execution time for the same three networks as function of the iteration.

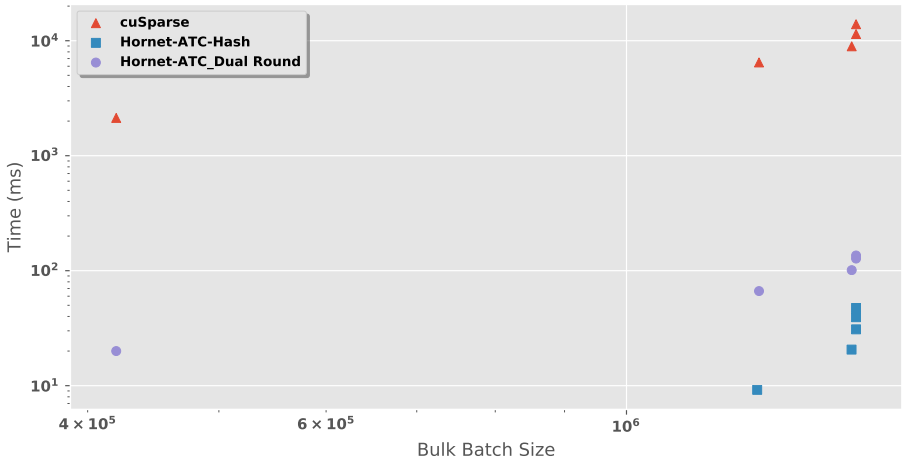



**(a)** Iterations as function of hash table size.

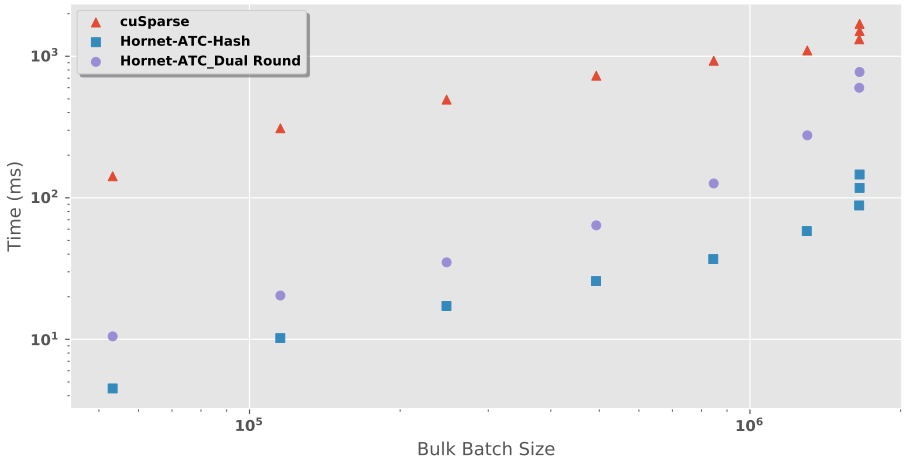



**(b)** Execution time (ms) as function of hash table size

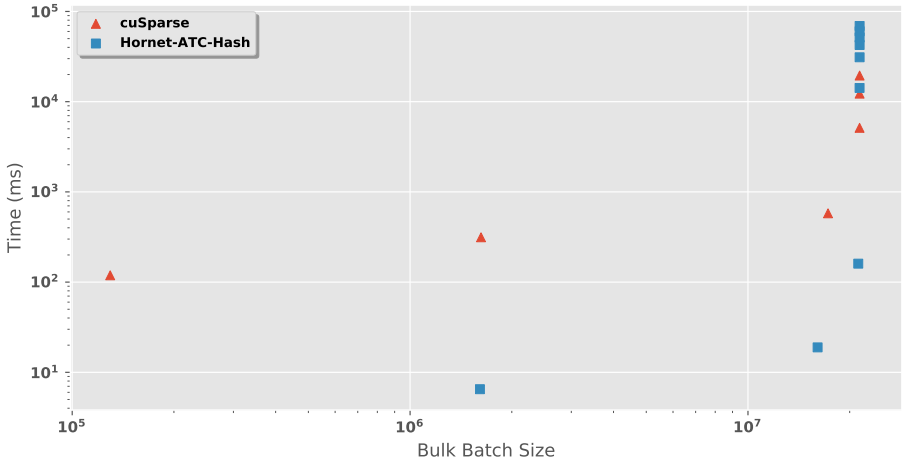
**Figure 6.5** The hash table size has an impact on both the number of iterations (top) and the execution (bottom). There is a clear correlation between these and selecting the right hash-table size is important for performance for denser transitive closure outputs.




**(a)** as-22July20




**(b)** delaunay-n13




**(c)** gnutella09

**Figure 6.6** Execution time of each phase as a function of the number of new edges found.

# CHAPTER 7

# CONCLUSION

In this thesis, we studied a novel algorithm for generating the transitive closure of a graph. The Anti-section transitive closure algorithm is intuitive in relation with the basic idea of transitive closure. The formulation can leverage the power of fine-grained parallelism and enables scalability. The method used to discover transitive edges called the Anti-Section operation is similar to list difference operation and finds edges efficiently. Moreover, through the analysis of performance of Hornet, we observe that redundant memory storage for graphs and operations to update the graph can be avoided by using a dynamic graph data structure saving time and space. Efficient batch-wise edge insertions is another advantage of Hornet that was highlighted through the performance analysis. In every computational phase of our algorithm, we take advantage of the parallel compute resources of the NVIDIA GPUs through parallelization. Graph sorting and updates are parallelized by functionalities Hornet provides and the Anti-Section operations are parallelized within our implementation. Ease in parallelization is introduced due to two major aspects. Distinction of generating transitive closure through distinct phases and the iteration-based formulation. These also contribute to the speedup of our algorithms over all sequential and parallel benchmarks.

## 7.1 Further Scope

For further efficiency, the Anti-Section operation can also be implemented using a binary search approach which might be more efficient in skipping edges that are not useful. Improvements can be achieved in memory management by estimating the size of the hash-table such that it achieves balance between the size of the hash-table and the number of iterations due to discovery of new edges in preceding ones. This can

be achieved by studying performance data for a large number of input graphs over varying sizes of hash-tables to devise a relation between a particular property of the graph and the most efficient hash-table size. A mathematical average case analysis can also be done for space and time complexity of both approaches. Improvements for denser graphs are possible in the context of hash-table size and Anti-Section operations.

# BIBLIOGRAPHY


[1] Rakesh Agrawal, Alexander Borgida, and Hosagrahar Visvesvaraya Jagadish. Efficient management of transitive relationships in large data and knowledge bases. *ACM SIGMOD Record*, 18(2):253–262, 1989.

[2] Rakesh Agrawal, Shaul Dar, and HV Jagadish. Direct transitive closure algorithms: Design and performance evaluation. *ACM Transactions on Database Systems (TODS)*, 15(3):427–458, 1990.

[3] Rakesh Agrawal and HV Jagadish. Direct algorithms for computing the transitive closure of database relations. In *VLDB*, volume 87, pages 1–4, 1987.

[4] Alfred V Aho. Data structures and algorithms, addison-wesley. *Reading, Mass.*, 1983.

[5] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[6] Vladimir L'vovich Arlazarov, Yefim A Dinitz, MA Kronrod, and IgorAleksandrovich Faradzhev. On economical construction of the transitive closure of an oriented graph. In *Doklady Akademii Nauk*, volume 194, pages 487–488. Russian Academy of Sciences, 1970.

[7] Francois Bancilhon. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems*, pages 165–178. Springer, 1986.

[8] Albert-László Barabási et al. *Network science*. Cambridge university press, 2016.

[9] Guy E Blelloch, Virginia Vassilevska, and Ryan Williams. A new combinatorial approach for sparse graph problems. In *International Colloquium on Automata, Languages, and Programming*, pages 108–120. Springer, 2008.

[10] Tommaso Bolognesi and Scott A Smolka. Fundamental results for the verification of observational equivalence: A survey. In *PSTV*, pages 165–179, 1987.

[11] Federico Busato, Oded Green, Nicola Bombieri, and David A Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on GPUs. In *2018 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[12] Søren Dahlgaard, Mathias Knudsen, and Mikkel Thorup. Practical hash functions for similarity estimation and dimensionality reduction. In *Advances in Neural Information Processing Systems*, pages 6615–6625, 2017.

[13] Timothy A Davis. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–25, 2019.

[14] Erik D Demaine, Felix Reidl, Peter Rossmanith, Fernando Sánchez Villaamil, Somnath Sikdar, and Blair D Sullivan. Structural sparsity of complex networks: Bounded expansion in random models and real-world graphs. *Journal of Computer and System Sciences*, 105:199–241, 2019.

[15] David Ediger, Rob McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5. IEEE, 2012.

[16] Michael J Fischer and Albert R Meyer. Boolean matrix multiplication and transitive closure. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 129–131. IEEE, 1971.

[17] J. Fox, O. Green, K. Gabert, X. An, and D. Bader. Fast and Adaptive List Intersections on the GPU. In *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2018.

[18] Fox, James and Tripathy, Alok and Green, Oded. Improving Scheduling for Irregular Applications with Logarithmic Radix Binning. In *IEEE Proc. High Performance Extreme Computing (HPEC)*, 2019.

[19] Thomas Gilray and Sidharth Kumar. Distributed relational algebra at scale. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 12–22. IEEE, 2019.

[20] O. Green, J. Fox, A. Tripathy, K. Gabert, E. Kim, X. An, K. Aatish, and D. Bader. Logarithmic radix binning and vectorized triangle counting. In *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2018.

[21] O. Green, P. Yalamanchili, and L.M. Munguía. Fast Triangle Counting on the GPU. In *IEEE Fourth Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–8, 2014.

[22] Oded Green and David A Bader. cuSTINGER: Supporting dynamic graph algorithms for gpus. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.

[23] Oded Green, Robert McColl, and David A Bader. GPU merge path: A GPU merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 331–340, 2012.

[24] Vesa Hirvisalo, Esko Nuutila, and Eljas Soisalon-Soininen. Transitive closure algorithm MEMTC and its performance analysis. *Discrete Applied Mathematics*, 110(1):77–84, 2001.

[25] Yang Hu, Hang Liu, and H Howie Huang. Tricore: Parallel triangle counting on GPUs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 171–182. IEEE, 2018.

[26] Yannis Ioannidis, Raghu Ramakrishnan, and Linda Winger. Transitive closure algorithms based on graph traversal. *ACM Transactions on Database Systems (TODS)*, 18(3):512–576, 1993.

[27] HV1093244 Jagadish. A compression technique to materialize transitive closure. *ACM Transactions on Database Systems (TODS)*, 15(4):558–598, 1990.

[28] Jan Jannink. Implementing deletion in B+-trees. *ACM Sigmod Record*, 24(1):33–38, 1995.

[29] Donald B Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.

[30] Gary J Katz and Joseph T Kider. All-pairs shortest-paths for large graphs on the GPU. 2008.

[31] James King, Thomas Gilray, Robert M Kirby, and Matthew Might. Dynamic sparse-matrix allocation on gpus. In *International Conference on High Performance Computing*, pages 61–80. Springer, 2016.

[32] Timothy G Mattson, Carl Yang, Scott McMillan, Aydin Buluç, and José E Moreira. GraphBLAS C API: Ideas for future versions of the specification. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2017.

[33] Ian Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[34] Esko Nuutila. An efficient transitive closure algorithm for cyclic digraphs. *Information Processing Letters*, 52(4):207–213, 1994.

[35] Esko Nuutila. Efficient transitive closure computation in large digraphs. 1998.

[36] Patrick E O'Neil and Elizabeth J O'Neil. A fast expected time algorithm for boolean matrix multiplication and transitive closure. *Information and Control*, 22(2):132–138, 1973.

[37] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. H-index: Hash-indexing for parallel triangle counting on GPUs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.

[38] Gerald Penn. Efficient transitive closure of sparse matrices over closed semirings. *Theoretical Computer Science*, 354(1):72–81, 2006.

[39] Paul Purdom. A transitive closure algorithm. *BIT Numerical Mathematics*, 10(1):76–94, 1970.

[40] Bernard Roy. Transitivité et connexité. *Comptes Rendus Hebdomadaires Des Seances De L Academie Des Sciences*, 249(2):216–218, 1959.

[41] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. GraphIn: An online high performance incremental graph processing framework. In *European Conference on Parallel Processing*, pages 319–333. Springer, 2016.

[42] Seppo Sippu and Eljas Soisalon-Soininen. A generalized transitive closure for relational queries. In *Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 325–332, 1988.

[43] Seppo Sippu and Eljas Soisalon-Soininen. *Languages and Parsing*. Springer, 1988.

[44] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[45] Patrick Valduriez and Setrag Khoshfian. Parallel evaluation of the transitive closure of a database relation. *International Journal of Parallel Programming*, 17(1):19–42, 1988.

[46] Henry S Warren Jr. A modification of warshall's algorithm for the transitive closure of binary relations. *Communications of the ACM*, 18(4):218–220, 1975.

[47] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.

[48] Martin Winter, Rhaleb Zayer, and Markus Steinberger. Autonomous, independent management of dynamic graphs on GPUs. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[49] Peter Zhang, Marcin Zalewski, Andrew Lumsdaine, Samantha Misurda, and Scott McMillan. GBTL-CUDA: Graph algorithms and primitives for GPUs. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 912–920. IEEE, 2016.